Radiology: Artificial Intelligence

EDITORIAL

Magician's Corner: 4. Image Segmentation with U-Net

Bradley J. Erickson, MD, PhD • *Jason Cai, MD*

From the Department of Radiology, Mayo Clinic, 200 First St SW, Rochester, MN 55905. Received September 12, 2019; revision requested October 30; revision received December 19; accepted December 23. **Address correspondence to** B.J.E. (e-mail: *bje@mayo.edu*).

Conflicts of interest are listed at the end of this article.

Radiology: Artificial Intelligence 2020; 2(1):e190161 • https://doi.org/10.1148/ryai.2020190161 • Content code: IN •

"A little magic can take you a long way."
Roald Dahl, *James and the Giant Peach*

Segmentation is the assignment of a label to pixels within an image and is a critical element of understanding an image (1,2). The label may identify an organ (eg, liver) or a pathologic type (tumor) and these labels are not necessarily mutually exclusive. Segmentation is an essential step for quantitative imaging (3), whether it be measuring a volume or a texture.

In prior articles, we used the FastAI *(https://www.fast.ai/)* deep learning framework, in part because we could achieve good results with very few lines of code. FastAI is built on top of PyTorch *(https://pytorch.org/)*, which is a framework developed by Facebook. TensorFlow *(https://www.tensorflow.org/)* is a machine learning framework developed by Google; it provides low-level access to many of the parameters that experts might wish to modify but is more difficult for beginners. Keras *(https://keras.io/)* is a framework that approaches network creation by having one line of Python code for each layer of a network, plus a few lines of code to set things up (eg, load data), and some lines to actually train the network. Keras originally used other libraries to do the computations, but more recently has become a part of TensorFlow.

In this article, we will use Keras to build a U-Net, which is a popular architecture for image segmentation (4). Up to this point, we have described the layers of a deep neural network only superficially. In this lesson, we will focus on the layers. As before, I encourage you to open the code for this article and run the cells as you read. First open the webpage *colab.research.google.com* and load the notebook (File > Open Notebook > Github > RSNA > MagiciansCorner > UNetWithTensorflow.ipynb). The first cell loads the Keras library and other libraries that we will be using. Execute cell 1 to set up the environment: in this case, we load TensorFlow libraries as well as keras/tensorflow libraries.

Cell 2 loads the data and prepares it for use. In this case, we are using abdominal CT data from 82 subjects from TCIA *(https://www.cancerimagingarchive.net/)*, in which the pancreas has been segmented. Only images that include the pancreas are included; abdomen soft-tissue window settings have been applied. Images have been converted to 8-bit gray-scale, cropped to include only the central 256 × 256 pixels, and compressed into four files. The next lines of code unzip (decompress) the images and separate them into training, validation, and test folders. The last lines of the cell load the unzipped and sorted image files into memory so they can be efficiently loaded into the GPU. Execute cell 2 if you haven't already.

Cell 3 defines the function for computing the Dice similarity coefficient (DSC). The DSC is 0 when the truth and predicted masks have no 1s that line up. Then because we want a loss function where lower numbers are good, we have a *dice_loss* function that is 1 – DSC. The other helper function in cell 3 does layer normalization. This will be described in a later article. Run cell 3 if you haven't already.

Cell 4 begins the task of training the algorithm to assign each pixel in each image with a label. As you learned in prior articles, machine learning requires a cost function—a numerical error measurement that we wish to minimize. When that error is at a minimum, then we have learned the task as well as we can. In the case of classification, the cost function was the classification error. In the case of segmentation, we often use the DSC (5). The DSC ranges from 0 (complete failure) to 1 (perfection) (Fig 1).

As with classification, we need training examples with the "right" answer as well as the prediction from our algorithm for each pixel. In this case, the "right" answer is the hand-traced "mask" for the pancreas. The mask has a value of "1" for each pixel where the pancreas is present on the image and is 0 elsewhere. If we define the set of traced pixels as X and the predicted mask as Y, we can calculate the DSC as

$$\mathrm{DSC} = \frac{2|X \cap Y|}{|X| + |Y|} \quad (1).$$

In computer code, we count the pixels where both X and Y are 1 and multiply that by 2 (the numerator in Equation [1]); we then divide that by the number of pixels that are "1" in X plus the number of "1" pixels in Y. Therefore, when there is perfect overlap of the truth mask and our predicted mask, X = Y, the DSC is 1.

Finally, we are ready to address our task of image segmentation. A very popular deep learning architecture for image segmentation is a U-Net (4). The main concept of the U-Net is that one repeatedly reduces the resolution of images (typically by four to five levels, each of which typically cuts resolution in half), then identifies the structures of interest, and then restores the resolution, while keeping track of the structures (Fig 2). Transposed convolution (sometimes incorrectly called *deconvolution* or *fractionally strided convolution*) is the technique we use to perform upsampling of an image with learnable parameters. Briefly, a transposed convolution is the application of a convolution function to an image where the pixels have had rows and columns inserted, effectively increasing the resolution. While one

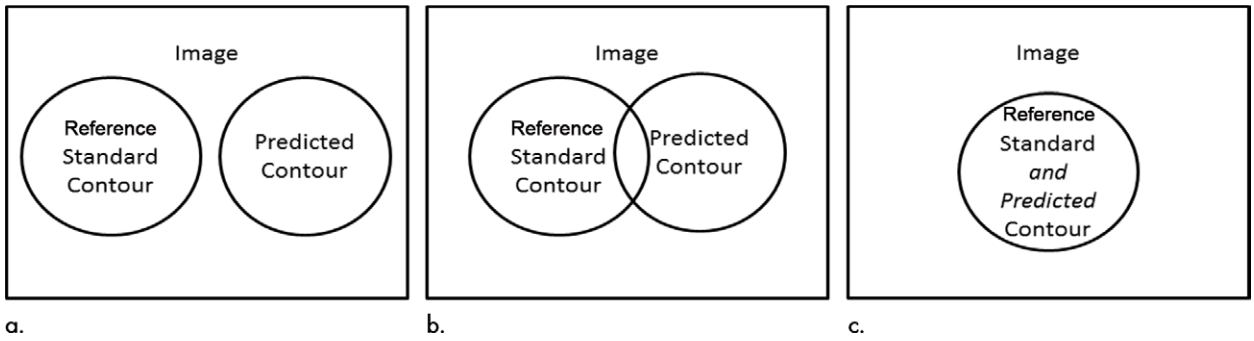

Figure 1: The Dice similarity coefficient (or Dice score) ranges from 0 to 1. **(a)** In the case where there is no overlap between the reference standard contour (such as provided by a human) and the predicted contour, the Dice score is 0. **(b)** In cases where there is a small overlap, the Dice score will be a small number, about 0.1 in this case. **(c)** In the case where they exactly match, the Dice score is 1.0.

could simply interpolate the values of the inserted pixels, transposed convolution takes such an image and applies convolution (where the kernel is part of the learning process) to produce a better upsampled image than simple interpolation would produce.

Now we focus in greater depth on the U-Net definition in cell 4. After declaring the function that will create the model (line 3), line 5 in the cell describes the image coming into the network. Next are the lines of code for convolution (two dimensional [2D] in our case), layer normalization, and pooling, as we go "down" the left side of the U-Net. Briefly, layer normalization helps to stabilize the learning process; adding it results in consistent improvement in performance with little computational penalty. After the multiple convolutions, we have found the key content needed to recognize the structure of interest. The right-hand side of the U-Net (transpose convolution and concatenation) restores the resolution of the image to provide more precise localization of the object boundaries.

Let us now investigate each layer more carefully. The first layer of the network is created by the line:

conv1 = Conv2D (8, 5, activation = 'linear', padding = 'same', kernel_initializer = init_fn) (inputs)

Conv2D means that a 2D convolutional layer is applied (see *https://keras.io/layers/convolutional/* for more complete documentation). The first parameter to this call ("8") refers to the number of "filters" that are created by this layer. A *filter* is the Keras term for a convolutional kernel. Having fewer filters means that fewer candidate kernels are kept and having too few will result in poor performance; having too many will waste memory and may also result in overfitting. The next parameter is the size of the convolution kernel (5 × 5 for the first layer; 3 × 3 for the rest). The next parameter is the activation function, and we use the popular rectified linear unit ('relu'). Padding is used to match the size of the input image and the output image. Finally, the layer has to start with some values for its weights (kernel values). Initialization is critical to success, and we use the initialization function described by He et al (6), hence the name *he_normal*. There are other initialization functions available *(https://keras.io/initializers/)*, and the user is encouraged to experiment with them.

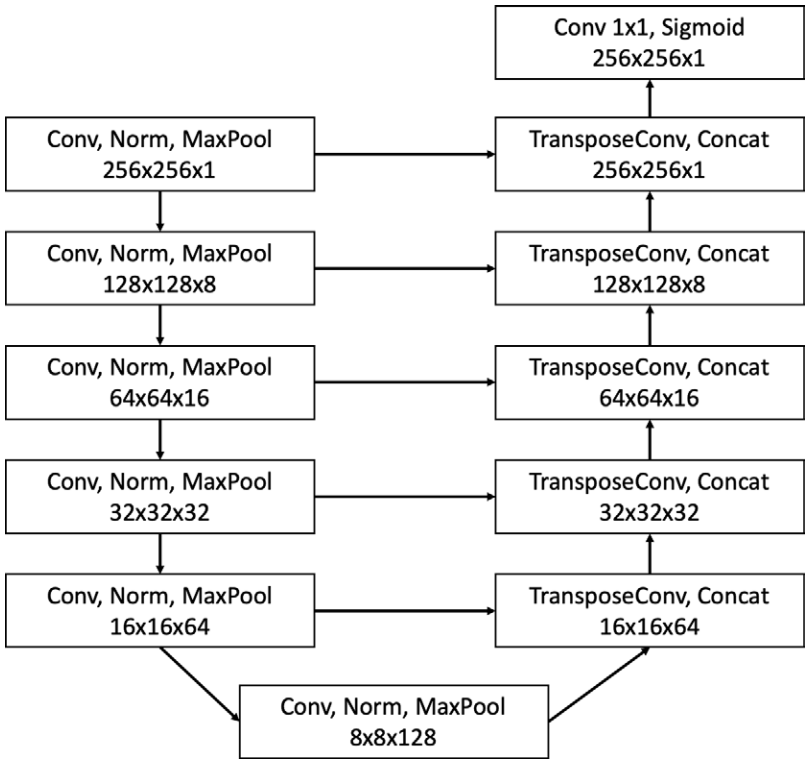

Figure 2: U-Net architecture. Each block on the left side consists of a convolution, layer-wise normalization, and maximum value pooling. The latter pools a 2 × 2 block into 1 pixel, thus reducing both X and Y dimensions by a factor of two. The blocks on the right side use transpose convolution with a stride of two to increase the X and Y dimensions by a factor of two. In addition, there is a skip connection where information from the down-resolution blocks is fed to these blocks. Conv = 2D convolution, Norm = layer-wise normalization, MaxPool = maximum value pooling, TransposeConv = transpose convolution, Concat = concatenation of convolution kernel from left block to the kernel in this block. Sigmoid is the function to map to a probability value from 0 to 1.

The next layer normalizes the output of the convolution to have a 0 mean and unit deviation. Recall this was defined in cell 3. We don't need to apply this after every convolution to get good results, but again, you are welcome to experiment with its use (and its removal).

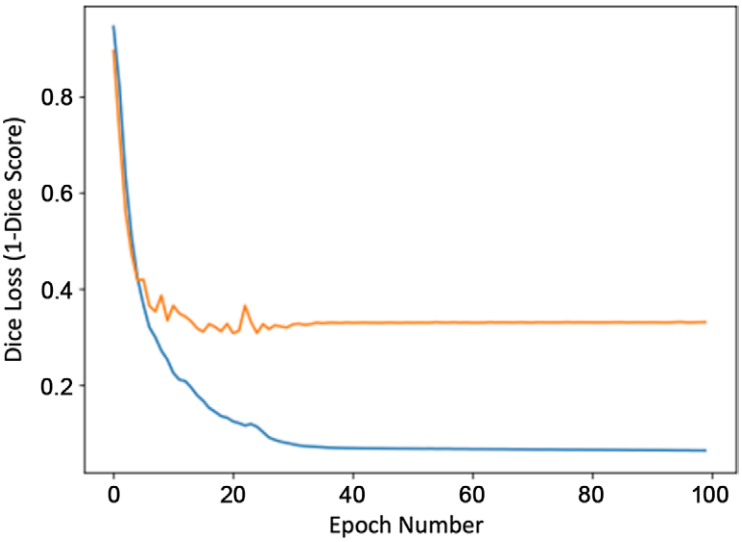

Figure 3: Graph shows the Dice loss (which is 1 − Dice similarity coefficient) at each epoch out to 100 epochs. Note that the blue and orange lines track each other until about 10, and after that point, the performance on the training set continues to improve but the performance on validation set plateaus, indicating that overfitting is occurring. Blue line = training loss curve, orange line = validation loss curve.

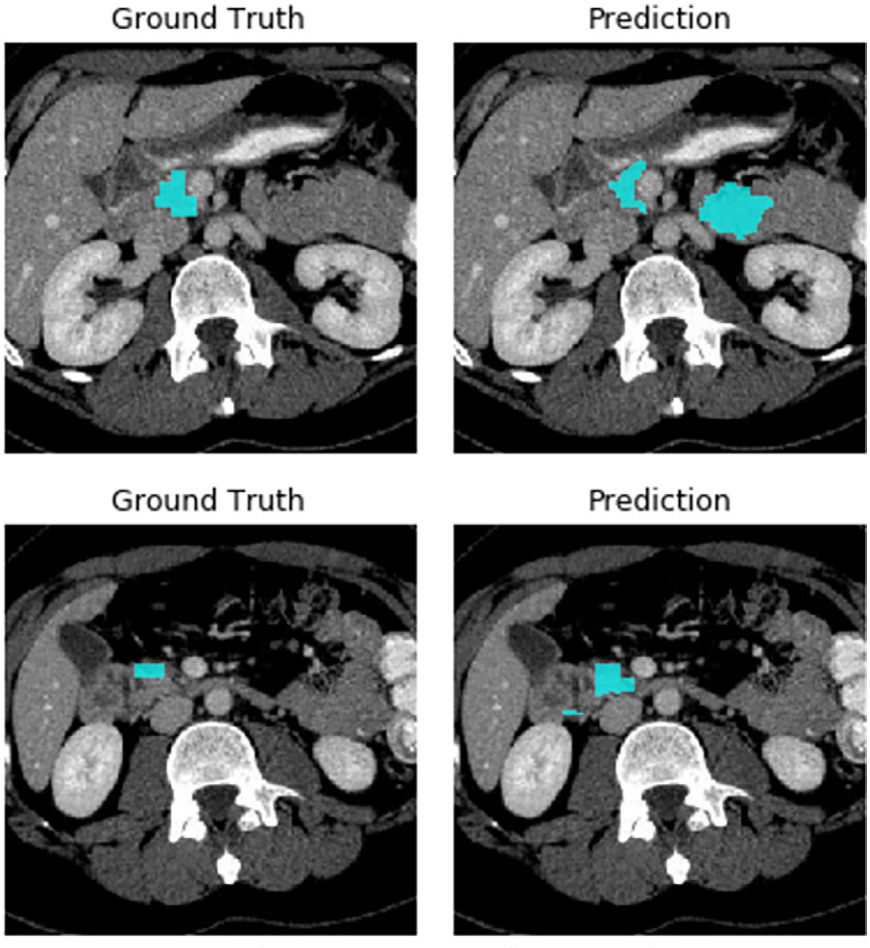

Figure 4: The performance on the test set of images. The images on the left are the ground truth (pancreas in cyan) and the right are the predicted. The overall Dice score for all images in the test set is 0.60, which is somewhat less than the validation performance but much less than the training set performance indicating that overfitting is occurring.

The next layer performs pooling, and as described before, MaxPooling simply replaces the "pool" with the maximum value. In this case, we are using a 2 × 2 pixel pool, so that means 4 pixels are replaced with 1, effectively cutting both the X and Y dimensions in half.

We now repeat these three steps (convolution, normalization, and pooling) for a total of five times, which means our image is now $256/2^5 = 8$ pixels on a side! But there is much more than the 8 × 8 image—we also have an array of 128 convolutional kernels that learn the best features in the image for identifying pancreas. The next steps use this information, as they restore the spatial resolution needed for precise localization of the boundaries. In the final step, a sigmoid function is used to constrain values to a range of 0 to 1. Note that softmax is the equivalent function when we have a multiclass problem.

The last line in the routine defines the optimizer to be used (Adam), the loss function (the Dice loss function defined in cell 3), and the metrics to be printed out for each epoch. Recall from prior articles that an epoch is defined as one complete pass through the training set. Note that because the GPU often does not have enough memory to load all examples at once, we divide the data into batches and when all the batches from the training set have been processed, that is one epoch. The Adam optimizer (7) will start with the learning rate (the "lr" parameter) supplied and will adjust it based on the improvement in the loss function; this choice is quite popular because it achieves good results with few epochs (8). Learning rate values between 1e-3 and 1e-5 tend to work well; starting with a larger learning rate (1e-3) may work best. Run cell 4, which will create the model.

Cell 5 does the work: it calls the routine to create an instance of the model developed in cell 4. The number of epochs and batch size are also set here. I ran this algorithm up to 100 epochs and the Dice score for the training set peaked at 0.97, which is quite good. However, performance on the validation and test peaked at 0.70, indicating overfitting (Fig 3). Running just 15 epochs had a lower validation performance (DSC ≈ 0.82) but the test set performance is nearly as good (DSC ≈ 0.69) and that is what you will find in the notebook you downloaded. But, as always, you are strongly encouraged to experiment!

In cell 6, we apply the trained network to our set of test cases. In the first line of cell 6, we load the model weights. We actually don't need this line, as the model is still available in memory, but in case you wish to save the weights and just do the

"inference" part at a different time, you can use this line to load the trained network. The next line applies the model ("predicts") to our set of test cases. This routine returns a vector of probabilities – 1 value for each test case. In the next line, we apply a threshold of 0.5 to determine whether it predicts a pixel as pancreas or not, and we convert that floating point value to 0 or 1. We next show a few images with the true segmentation (left side) and the predicted segmentation (right side) (Fig 4). Last, we calculate the actual DSC for all the test images using a function that computes the DSC for each image, and then computes the average (our Dice function in cell 3 returns an array of values).

Clearly the network is generally finding the pancreas, but just as clearly, it is also making mistakes: it both selects pixels far from the expected location of the pancreas, as well as not identifying pixels that are pancreas. This demonstrates the reason that deep learning is not trivial. To improve performance, one can try more training or more complex architectures. I encourage the reader to try applying this approach to relatively simple segmentation tasks (eg, liver, spleen, brain) and to experiment with some of the parameters in the network.

In this article, the Keras framework was used to create a U-Net that could successfully identify the pancreas in abdomen CT images. Specific components of the U-Net were described, including 2D convolution and its counterpart, transpose convolution. We also applied layer-wise normalization and max pooling. We created a custom loss function based on the DSC. These are the basic components of the U-Net, which is a popular segmentation method.

Author contributions: Guarantors of integrity of entire study, B.J.E., J.C.; study concepts/study design or data acquisition or data analysis/interpretation, B.J.E., J.C.; manuscript drafting or manuscript revision for important intellectual content, B.J.E., J.C.; approval of final version of submitted manuscript, B.J.E., J.C.; agrees to ensure any questions related to the work are appropriately resolved, B.J.E., J.C.; literature research, B.J.E., J.C.; experimental studies, J.C.; and manuscript editing, B.J.E., J.C.

Disclosures of Conflicts of Interest: B.J.E. disclosed no relevant relationships. **J.C.** Activities related to the present article: disclosed no relevant relationships. Activities not related to the present article: employed by Mayo Clinic (salary for research fellow position). Other relationships: disclosed no relevant relationships.

References

1. Sharma N, Aggarwal LM. Automated medical image segmentation techniques. J Med Phys 2010;35(1):3–14.
2. Suetens P, Bellon E, Vandermeulen D, et al. Image segmentation: methods and applications in diagnostic radiology and nuclear medicine. Eur J Radiol 1993;17(1):14–21.
3. Sullivan DC, Obuchowski NA, Kessler LG, et al. Metrology Standards for Quantitative Imaging Biomarkers. Radiology 2015;277(3):813–825.
4. Ronneberger O, Fischer P, Brox T. U-Net: Convolutional Networks for Biomedical Image Segmentation. In: Navab N, Hornegger J, Wells WM, Frangi AF, eds. Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. MICCAI 2015. Lecture Notes in Computer Science, vol 9351. Cham, Switzerland: Springer, 2015; 234–241.
5. Sørensen R. Temperatur- og pulsforhold ved appendicitis belyst ved 2.250 tilfaelde. Nord Med 1948;40(51):2389.
6. He K, Zhang X, Ren S, Sun J. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. arXiv [cs.CV]. [preprint] http://arxiv.org/abs/1502.01852. Posted 2015. Accessed June 12, 2019.
7. Kingma DP, Ba J. Adam: A Method for Stochastic Optimization. arXiv [cs.LG]. [preprint] http://arxiv.org/abs/1412.6980. Posted 2014. Accessed October 12, 2018.
8. Optimizers - Keras Documentation. https://keras.io/optimizers/. Accessed September 3, 2019.